Design Recipe for Methods

(credit to Stephen Bloch of Adelphi University, David Kay of UC Irvine, and Therapon Skonteiniotis of Northeastern University; much of this handout uses their work)

Design Recipe in a nutshell

1. Develop data definitions and data examples
2. Define the purpose statement
3. Write the contract
4. Write examples of how you would you use this method
5. Draft a template
6. Write the body
7. Run the tests
8. Review/Iterate

Using the Design Recipe to write methods

Every time you write methods you need to follow these steps of the Design Recipe. By following the Design Recipe, you
will approach solving problems in an organized and systematic way. You will avoid making common mistakes. You will have
a much stronger grasp of the problem and how to solve it. You will avoid “blank screen syndrome”; otherwise known as
"I-understand-the-problem-but-I’m-stuck-and-I-can’t-get-started". Each step of the design recipe will be explained; we
will use the following problem statement to demonstrate how each step is used: Write a problem that calculates any power
of any integer.

Step 1: Develop data defintions and data examples

What is the input? What are the parameters to your method? What is the name of the parameters? What is the data type of your
parameters? Are there any restrictions on the range or quality of each input?

In our example problem statement, we need to calculate the power of any integer. What inputs do we need? From math, we know that
we need some `base` that will be multiplied ‘x’ number of times where ‘x’ is the exponent. Thus, our problem requires two
inputs: `base` and `exponent`. We have names for this data but what are the data types? Well, for our purposes, let’s limit
`base` to be a data type of `int`. For our exponent, let’s keep things simple and set the data type to also be an `int`.

Now that we have defined our inputs, what about our outputs? What's the output? Where does it go? Is it returned from the method?
Is it printed? Does it write the results to a file? If we do return something, what are we returning? What is the data type that
we are returning?

In our example, we need to return the result of taking a base and raising it to an exponent. What should we return? A number of course!
But in Java, what data type can we use to return the product of a bunch of numbers? We can use a `double`, `float`, `long`, or an `int`.
For our problem, let's keep it simple and return an `int`.

Step 2: Define the purpose statement

Write a one-line purpose statement that describes in plain English what the method does.

In our example, we can write its purpose statement as the following: "Calculates the result of raising the base to the power of the exponent."

Step 3: Write the contract

Write a contract that states the method's name, the names and types of its parameters, and the type of its return value in pseudocode.
Using our example, our contract can be written as the following: `getPower(int base, int power) => int`

Using Java, we can translate our contract into real code by creating a method header. For example:

```
public static int getPower(int base, int power) {
...
}
```

Some rules of thumb:

- The "type of information it returns" in the contract becomes the return type (or "void" if nothing is returned)
- The method name becomes the method name (duh!)
- For each piece of "information needed" put the type and parameter name inside the paranthesis
- Always write both the starting and ending braces of the method definition so that you don't forget that you still have to write the body of the code

Step 4: Write examples of how you would use this method

Write down one or more examples of how you would use this method, along with the "right answer" you expect it to produce. This step is useful in at least three ways.

- It allows you to try out the syntax people will use to invoke the method, and perhaps learn that your first guess at the syntax is inconvenient to use, so you should change the syntax before wasting a lot of time implementing it.

- It allows you to discover, early on, that your contract wasn't quite right, and that you actually want different kinds of information going in or out. If that happens, change the contract now and re-write your examples accordingly; that will waste less time than going on and having to change the contract later.
- It gives you a ready source of test cases, already in legal Java syntax.

Start with simple examples and work up to more complicated ones; put them in the `main` method. Again, using our problem of writing
a method that calculates the power of any base, we write our examples as seen in the following:

```
public static void main(String[] args) {
  System.out.println("getPower(2,4) = " + getPower(2, 4) + " (should be 16).");
  System.out.println("getPower(-3,3) = " + getPower(-3, 3) + " (should be -27)."
  System.out.println("getPower(-3, 2) = " + getPower(-3, 2) + " (should be 9).")
  System.out.println("getPower(73748, 0) = " + getPower(73748, 0) + " (should be
  System.out.println("getPower(0, 46) = " + getPower(0, 46) + " (should be 0).")
}
```

Step 5: Draft a template

You actually have somewhat more information at this point than what will fit into the header, and you can use this to fill in some of the body.

For example, if your method is supposed to return a String, you can be pretty sure that the last statement in its body will be
return something-of-type-String;
You just need to figure out what the something-of-type-String is, and you may be done with the body.

Again, using our problem of writing a method that calculates the power of any base, we can draft up the template as seen in the following code:

```
public static int getPower(int base, int power) {
    ....
    return something-of-type-int;
}
```

Step 6: Write the body

Write the body of your method. In other words, write some code! Finally! Fill in the stuff in between the braces in the method definition. This is the part that takes the most thinking, but we've tried to avoid "blank-page syndrome" by writing the contract, examples, header, and template first. By doing all prep work in steps 1-5, we can start writing some code without being too stuck staring at a blank screen.

Using our example, here's one way of writing the body:

```
public static int getPower(int base, int power) {
    int result = 1;
    for (int times = 1; times <= power; times++) {
      result = result * base;
    }
```

```
    return result;
}
```

Step 7: Run the tests

Testing is essential if you want to have any faith that your programs work. If you can't be bothered to test your programs, I can't be bothered to grade them. Fortunately, we have already written our tests in Step 4. To run our test, we just simply need to run the main method in our program. The complete
code of our example is shown below:

```
public class Example {
  public static void main(String[] args) {
    System.out.println("getPower(2,4) = " + getPower(2, 4) + " (should be 16).")
    System.out.println("getPower(-3,3) = " + getPower(-3, 3) + " (should be -27)
    System.out.println("getPower(-3, 2) = " + getPower(-3, 2) + " (should be 9).
    System.out.println("getPower(73748, 0) = " + getPower(73748, 0) + " (should
    System.out.println("getPower(0, 46) = " + getPower(0, 46) + " (should be 0).
  }

  public static int getPower(int base, int power) {
    int result = 1;
    for (int times = 1; times <= power; times++) {
      result = result * base;
    }

    return result;
  }

}
```

Compile your program. If it has syntax errors, read the error messages carefully, figure out what they mean, fix them, and compile again until there are no syntax errors.

For each of your test cases, observe the actual output. If it's not what you expected, either you were expecting the wrong thing (possible, though unlikely) or there's a bug in your program. Figure out how the answer is wrong (not just that it's wrong) and how this wrong answer could have happened, fix it, and compile and run again until all test cases produce correct answers. Remember, whenever you fix a bug, be sure to re-test the previously working examples to make sure they still work!

Step 8: Review and Iterate

Did all your tests pass in Step 7? If not, then you need to review all your previous steps and see where your error was. If your tests did pass, try adding more tests to see how robust your method is to many different kinds of inputs. Try to "break" your method by calling it with a lot of different
inputs and validating the expected output.

Review your implementation (i.e. the body of your method). Could you have written the code clearer?